TUTOR

The Truth About Free System Resources

BY JEFF PROSISE

Are there certain Windows applications that you just can't live without? If there are, you probably set up your system so that it would run Windows well. You probably upgraded the RAM in your PC and configured a swap file on your hard disk in order to take advantage of virtual memory. Then, after loading three or four applications, you discovered that Windows wouldn't let you load any more programs. Program Manager says that there are still several megabytes of RAM available, but to load another program, you have to close one of the active ones first.

What's going on here? You just .rned the hard way that available RAM isn't the only factor that limits the number of programs you can load concurrently in Windows. A more unavoidable constraint is the amount of free system resources. In simple terms, *system resources* refers to a limited area of memory that Windows sets aside internally to store data about windows and objects created by Windows programs, and *free system resources* (FSR) is the percentage of this memory that isn't being used.

You can find out what your FSR percentage is from Program Manager's About box. Loading a large application such as Word for Windows or Excel generally uses up 5 to 10 percent of your FSR in Windows 3.1, and often more in Windows 3.0. No matter how much RAM your PC contains, Windows will refuse to load more programs when FSR approaches zero. Furthermore, it's wise never to let the FSR percentage drop too low. All sorts of strange things may happen when FSR goes below 10 or 15 percent, ranging from the benign—for exam-. icons losing their titles—to full-ıedged system crashes.

FSR is something every Windows user should be aware of, but it is one of the aspects of Windows that isn't widely understood. In this installment of Tutor, you'll learn about system resources, how Windows handles them, and how that handling differs in 3.0 and 3.1. You'll also acquire a useful new utility for keeping tabs on FSR, complete with source code.

FREE SYSTEM RESOURCES DEFINED Windows relies extensively on *data structures*—collections of variables that are grouped together because they serve a related purpose. Many of these structures are not static, but are created and destroyed on-the-fly as circumstances require. For example, when you open a window, Windows creates a data structure in memory that belongs to Windows, and that data structure contains important information such as who the window's owner is, the window's size, and its position on the screen. Many windows contain other windows (controls such as buttons and list boxes are actually windows), so a single application program frequently begets several window data structures.

Get savvy about Windows' free system resources: Here's a clear explanation and a utility that lets you keep track of them.

The window data structure is only one example of the many data structures that Windows uses internally. Many of these internal data structures were never intended to be accessed directly and are therefore not documented. We only know about them through inference and through texts like *Undocumented Windows* by Andrew Schulman and others (1992, Addison-Wesley). Other items stored this way include menus, window classes, device contexts, and GDI (Graphics Device Interface) objects such as brushes, pens, bitmaps, and fonts. Each data structure that Windows creates consumes a small amount of memory. Free system resources is the percentage of memory that Windows sets aside to hold internal data structures that are currently unused.

Where does Windows find room to store the data structures? In the local heaps of two of the program modules (actually DLLs) that make up Windows—GDI.EXE and USER.EXE. In Windows, every program is assigned at least one data segment whose length is 64K or less. Part of the data segment is used to store variables created by the program, and part of it is used to hold the program's stack. The remainder of the segment—the free memory—constitutes the program's *local heap*. If desired, a program may allocate memory from its local heap by calling a Windows API function called LocalAlloc. GDI and USER contain their own local heaps, and it is inside these heaps that Windows stores the aforementioned data structures. These heaps are the *system resources* in *free system resources*.

To understand the mechanism by which system resources are consumed, consider what happens when a program creates a GDI brush. First GDI calls LocalAlloc to acquire enough heap space to hold a brush data structure. Then it initializes the data structure and passes the address back to the program that created the brush to use as a *handle*—a number that uniquely identifies the brush. The free space in GDI's heap diminishes by the number of bytes required for the brush data structure. Given that there may be dozens (more likely, hundreds) of GDI objects defined in the system at any time, and that every one of them consumes space in GDI's heap, it's easy to imagine how the heap could fill up very quickly.

In Windows 3.0, USER contained two local heaps and GDI contained one. One USER heap stored data structures for windows, menus, window classes, and other USER objects; the other stored Window's global atom table (a systemwide "dictionary" to which programs can add and delete text strings). The GDI heap stored all data structures related to Windows' Graphics Device Interface. By analyzing Windows 3.0 running real-world application programs, Microsoft determined that menus were causing the greatest strain on FSR, so in Version 3.1 they added two more heaps to USER—one to hold menu data structures and one to hold menu strings. GDI also gained an extra local heap in 3.1.

Earlier, we defined free system resources as the amount of memory remaining for Windows to store internal data structures. We can now refine this definition to say that in Windows 3.0, FSR is the percentage of free space remaining in the primary USER heap or the percentage of free space remaining in the GDI heap, whichever is less. In 3.0, USER typically ran out of heap space long before GDI, so GDI rarely factored into FSR. It is not entirely clear how Windows 3.1 computes FSR. It probably divides total free space in the USER heaps by the combined length of the USER heaps, does the same for GDI, and reports the lesser of the two. It also appears that Program Manager doesn't use the GetFreeSystemResources API function, which debuted in Windows 3.1, because the percentage reported by the function and the one reported by Program Manager occasionally disagree by a percent.

The good news to users is that because of the increased heap space in USER and GDI, Windows 3.1 is less susceptible to FSR shortages than 3.0. But limited FSR is still a problem, and it is likely to continue to be until we move to a 32-bit oper-

FSR.C
1 of 2

```
/**********************************************************************
  FSR displays the percentage of free system resources in real time.
  Copyright (c) 1993 Jeff Prosise. First published in PC Magazine,
  U.S. Edition, March 16, 1993.
 **********************************************************************/

#include <windows.h>
#include <stdlib.h>

#define WARNINGLEVEL 20
#define IDM_ALWAYSONTOP 100

extern DWORD FAR PASCAL GetHeapSpaces (HMODULE);
long FAR PASCAL WndProc (HWND, WORD, WORD, LONG);
DWORD GetFSR (BOOL);

WORD wTimerID;                          // Timer ID

/**********************************************************************
  WinMain starts the program.
 **********************************************************************/

int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
                LPSTR lpszCmdLine, int nCmdShow)
{
    static char szAppName[] = "Free System Resources";
    WNDCLASS wndclass;
    HWND hwnd;
    MSG msg;

    if (hPrevInstance)
        return FALSE;

    wndclass.style = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc = (WNDPROC) WndProc;
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.hInstance = hInstance;
    wndclass.hIcon = NULL;
    wndclass.hCursor = LoadCursor (NULL, IDC_ARROW);
    wndclass.hbrBackground = GetStockObject (LTGRAY_BRUSH);
    wndclass.lpszMenuName = NULL;
    wndclass.lpszClassName = szAppName;

    RegisterClass (&wndclass);

    hwnd = CreateWindow (szAppName, szAppName, WS_OVERLAPPEDWINDOW,
            CW_USEDEFAULT, CW_USEDEFAULT, 384, 96, NULL, NULL,
            hInstance, NULL);

    if ((wTimerID = SetTimer (hwnd, 1, 1000, NULL)) == NULL) {
        MessageBox (hwnd, "Unable to allocate a timer", "Error",
            MB_ICONEXCLAMATION | MB_OK);
        return FALSE;
    }

    ShowWindow (hwnd, nCmdShow);
    UpdateWindow (hwnd);

    while (GetMessage (&msg, NULL, 0, 0))
            DispatchMessage (&msg);

    return msg.wParam;
}

/**********************************************************************
  WndProc processes messages to the main window.
 **********************************************************************/

long FAR PASCAL WndProc (HWND hwnd, WORD message, WORD wParam, LONG lParam)
{
    static char szSection[] = "Options";
    static char szEntry[] = "AlwaysOnTop";
    static char szIniFile[] = "FSR.INI";

    static HPEN hGrayPen;
    static HBRUSH hPinkBrush, hRedBrush;
    static DWORD dwPercentFree, dwOldPercent;
    RECT rect, rectUsed, rectFree;
    DWORD dwVersion, dwTotal;
    static BOOL bWin31Flag;
    static HMENU hSysMenu;
    char szBuffer[4];
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message) {

    case WM_CREATE:
        dwVersion = GetVersion ();
        if ((LOBYTE (LOWORD (dwVersion)) >= 3) &&
            (HIBYTE (LOWORD (dwVersion)) >= 10))
            bWin31Flag = TRUE;
        else
            bWin31Flag = FALSE;

        hGrayPen = CreatePen (PS_SOLID, 1, RGB (128, 128, 128));
        hPinkBrush = CreateSolidBrush (RGB (192, 0, 192));
        hRedBrush = CreateSolidBrush (RGB (192, 0, 0));

        dwPercentFree = GetFSR (bWin31Flag);
        dwOldPercent = dwPercentFree;

        if (bWin31Flag) {
            hSysMenu = GetSystemMenu (hwnd, FALSE);
            AppendMenu (hSysMenu, MF_SEPARATOR, 0, NULL);
            AppendMenu (hSysMenu, MF_STRING, IDM_ALWAYSONTOP,
                "Always on &Top");
            if (GetPrivateProfileInt (szSection, szEntry, 0, szIniFile)) {
                CheckMenuItem (hSysMenu, IDM_ALWAYSONTOP, MF_CHECKED);
                SetWindowPos (hwnd, HWND_TOPMOST, 0, 0, 0, 0,
                    SWP_NOMOVE | SWP_NOSIZE);
            }
        }
        return 0;

    case WM_PAINT:
        hdc = BeginPaint (hwnd, &ps);
        SetBkMode (hdc, TRANSPARENT);
        GetClientRect (hwnd, &rect);

        if (IsIconic (hwnd)) {
            if (dwPercentFree >= WARNINGLEVEL) {
                FillRect (hdc, &rect, GetStockObject (LTGRAY_BRUSH));
                SetTextColor (hdc, RGB (0, 0, 192));
            }
```

***Figure 1:* The source code for FSR.EXE, a Windows utility that tracks free system resources in real time.**

FSR.C

2 of 2

```
        else {
            FillRect (hdc, &rect, hRedBrush);
            SetTextColor (hdc, RGB (255, 255, 255));
        }
        DrawText (hdc, _itoa ((int) dwPercentFree, szBuffer, 10),
            -1, &rect, DT_SINGLELINE ¦ DT_CENTER ¦ DT_VCENTER);
    }

    else if ((--rect.bottom < 16) ¦¦ (--rect.right < 32))
        FillRect (hdc, &rect, GetStockObject (LTGRAY_BRUSH));

    else {
        InflateRect (&rect, -4, -4);

        SelectObject (hdc, hGrayPen);
        MoveTo (hdc, rect.left, rect.bottom);
        LineTo (hdc, rect.left, rect.top);
        LineTo (hdc, rect.right+1, rect.top);
        MoveTo (hdc, rect.right-1, rect.top+1);
        LineTo (hdc, rect.left+1, rect.top+1);
        LineTo (hdc, rect.left+1, rect.bottom);

        SelectObject (hdc, GetStockObject (WHITE_PEN));
        MoveTo (hdc, rect.left+1, rect.bottom);
        LineTo (hdc, rect.right, rect.bottom);
        LineTo (hdc, rect.right, rect.top);
        MoveTo (hdc, rect.right-1, rect.top+2);
        LineTo (hdc, rect.right-1, rect.bottom-1);
        LineTo (hdc, rect.left+1, rect.bottom-1);

        InflateRect (&rect, -2, -2);
        rect.right++;
        rect.bottom++;
        CopyRect (&rectUsed, &rect);
```

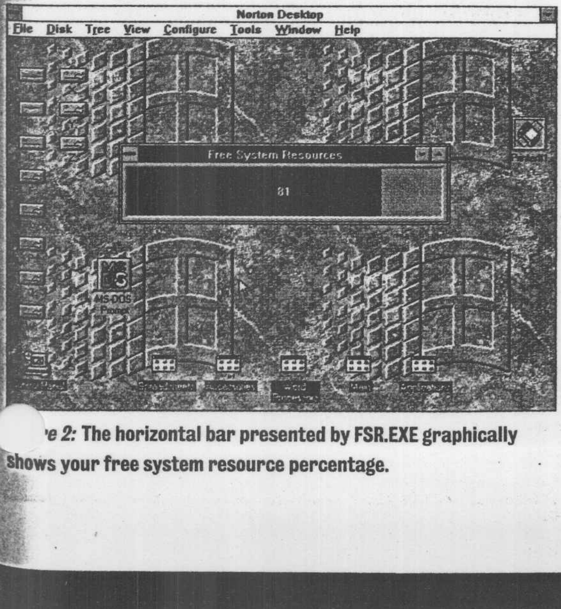
```
            CopyRect (&rectFree, &rect);
            dwTotal = (DWORD) (rect.right - rect.left);
            rectFree.right = rectFree.left +
                ((int) ((dwTotal * dw          DeleteObject (hRedBrush);
        DeleteObject (hGrayPen);
        PostQuitMessage (0);
        return 0;
    }
    return DefWindowProc (hwnd, message, wParam, lParam);
}

/*******************************************************************
  GetFSR returns a value between 0 and 100 that quantifies the
  percentage of free system resources.
 *******************************************************************/

DWORD GetFSR (BOOL bWin31)
{
    DWORD dwHeapSpaces;
    WORD wUSERPercentFree, wGDIPercentFree;

    if (bWin31)
        return ((DWORD) GetFreeSystemResources (GFSR_SYSTEMRESOURCES));
    else {
        dwHeapSpaces = GetHeapSpaces (GetModuleHandle ("USER"));
        wUSERPercentFree = (WORD) (((DWORD) (LOWORD (dwHeapSpaces)) * 100) /
            ((DWORD) (HIWORD (dwHeapSpaces))));
        dwHeapSpaces = GetHeapSpaces (GetModuleHandle ("GDI"));
        wGDIPercentFree = (WORD) (((DWORD) (LOWORD (dwHeapSpaces)) * 100) /
            ((DWORD) (HIWORD (dwHeapSpaces))));
        return ((DWORD) min (wUSERPercentFree, wGDIPercentFree));
    }
}
```

ating system such as Windows NT or OS/2. In 16-bit Windows, increasing the heap space would mean creating more local heaps or moving internal data structures to the global heap. The former is esirable because it would make Win- ws as a whole more fragmented (that is, the data structures would be divided among more heaps) and still not solve the problem (if one local heap fills up, you may still run out of FSR even if the other heaps are nearly empty). The latter would bog down performance because of the segment register manipulations required. A 32-bit operating system makes the issue of local versus global heaps moot, because the 386 and 486 do not suffer from 64K segmentation limits.



e 2: The horizontal bar presented by FSR.EXE graphically shows your free system resource percentage.

GAUGING FREE SYSTEM RESOURCES Figure 1 lists the C source code for a Windows utility named FSR.EXE, which tracks free system resources in real time. When it is displayed as a window, FSR draws a horizontal bar that represents the percentage of free system resources, as shown in Figure 2. The bar scales to the size of the window, so you can make it as large or as small as you'd like. If free system resources fall below 20 percent, the bar changes from magenta to red as a warning. When it is reduced to an icon, FSR displays the percentage of free system resources in numeric form. The display is updated once a second. With FSR running, you should never again have to open Program Manager just to check free system resources.

The heart of FSR is the function named GetFSR, which returns a DWORD (double word) value equal to the percentage of system resources free. In Windows 3.1, GetFSR calls the documented GetFreeSystemResources function to get a count of free system resources. In Windows 3.0, it calls the undocumented GetHeapSpaces function, which accepts a module handle and returns a DWORD whose high word is equal to the number of bytes in the default local heap and whose low word is equal to the number of free bytes in the heap. GetFSR calls GetHeapSpaces once with the module handle of USER and once with the module handle of GDI, computes percent free in each module by dividing free bytes by total bytes and multiplying by 100, and then returns the lesser of the two percentages. GetHeapSpaces was discussed in *PC Magazine*'s November 12, 1991, Windows column, and it is treated at length in *Undocumented Windows.* GetHeapSpaces produces erroneous results in Windows 3.1, because it does not take into account the additional heaps set aside for USER and GDI.

If you run it under Windows 3.1, FSR adds an "Always on Top" option to its system menu. When checked, "Always on Top" ensures that the FSR window stays visible even if another application is running full-screen. The program remembers how you last set it and automatically assumes the same configuration the next time you start it. You can download the FSR executable and related files from the Tutor section of the Utilities/Tips Forum on PC MagNet. FSR.EXE is the executable only; FSR.ZIP contains FSR.EXE, a document file, the source code, and the other files you need to compile FSR.EXE with Microsoft C 7.0 and Windows 3.1's SDK. □